# DESIGN, CONFORMANCE VERIFICATION, AND PERFORMANCE EVALUATION OF OPENFLOW MESSAGE LAYER

A Thesis

by

DAOQI WANG

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

| | |
|---|---|
| Chair of Committee, | Alex Sprintson |
| Committee Members, | Narasimha Reddy |
| | Ana Goulart |
| | Committee Member3 |
| Head of Department, | Chanan Singh |

May 2014

Major Subject: Computer Engineering

# ABSTRACT

The message layer is a critical part of a network protocol stack. The quality of the protocol implementation is closely tied to that of the message layer. The responsibilities of the message layer include conversion between binaries from the network interface and message structures in the device memory, along with validation of these messages. Design and implementation of the message layer pose major challenges, especially for modern protocols that have complex message structures. Poor design choices and errors in the implementation lead to safety issues, performance inefficiencies, and expose vulnerabilities, opening door to potential exploitations and attacks.

In this thesis we develop a systematic approach to design and implementation of efficient and correct-by-construction message layer components of the protocol stack. To achieve this goal, we identify some common design trade-offs and evaluate their impacts on performance and/or safety using the OpenFlow protocol as a case study. A performance benchmarking framework leveraging existing tools is developed to conduct these evaluations. Furthermore, the thesis develops a framework for conformance verification. The conformance framework proposes a methodology that generates test messages to identify vulnerabilities in the message layer. In particular, we present an algorithm that minimizes the number of required test messages by exploiting the structure of the message format.

A safe message layer is developed as part of the Flowgrammmable OpenFlow stack. Flowgrammable and several existing OpenFlow stack implementations, such as Beacon Controller and CPqD Soft Switch, are evaluated using both proposed conformance and performance frameworks. It is shown that unlike Flowgrammable, implementations of Beacon and CPqD Soft Switch message layer contain confor-

mance violations. Furthermore, design choices such as omitting semantic checking, inlining small and frequently used functions, and header optimization can yield performance gain in protocol stack implementations. The work can be extended by automating the message layer implementation and testing by using a programming language approach. Even though OpenFlow is used as an example, the work can be applied to improve the performance and level of conformance in other network protocols as well. This thesis intends to help developers take a more systematic approach and make better design choices in implementing a conformant message layer for a broad range of network protocols.

DEDICATION

To my family and friends

# ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Dr. Alex Sprintson, for all the mentoring he has given me in this work and throughout my entire graduate school experience. During this period, I have grown up for a tremendous amount both in my technical aptitude and as a person. I would also like to sincerely thank Jasson Casey and Dr. Andrew Sutton for their advice, along with the opportunity to participate in Flowgrammable as a major open source contributor. I am very grateful for the help I received from my colleagues from Flowgrammable Colton Chojnacki, Atin Ruia, Muxi Yan, and Allen Webb for this work. Finally, I want to thank Drs. Narasimha Reddy and Ana Goulart for having served as my committee members.

# NOMENCLATURE

| | |
|---|---|
| LTE | Long Term Evolution |
| NIC | Network Interface Card |
| TCP | Transmission Control Protocol |
| UDP | User Datagram Protocol |
| TLS | Transport Layer Security |
| ASCII | American Standard Code for Information Interchange |
| HTTP | Hypertext Transfer Protocol |
| IPC | Instruction per Cycle |

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

# LIST OF TABLES

TABLE Page

# 1. INTRODUCTION

Today's computing and networking systems are characterized by increasing complexity, multiple administrative domains, diversity of basic components, and increasing uncertainty about their underlying topology and internal structure, cross-layer interactions, and administration policies. At the same time, there is a competitive pressure to quickly deploy large-scale networking systems that use a large number of network protocols.

Protocols are an essential element any networking system. In recent years, there is an explosion in the number of network protocols for both wireless and wireline communications. The protocols are being implemented a wide range of devices, such as smartphone, personal computers and high end servers. Many protocol stacks are being implemented by both commercial and open source communities.

Different protocol stacks have different goals. For example, the protocol stacks developed for the servers need to maximize the availability whereas the protocol stacks developed for smartphones need to be power efficient. The complexity of the protocols has been increasing as well. New protocols such as LTE needed to be defined by lengthy specification. A complete protocol stack includes many different protocols spanning multiple layers, resulting in a even higher complexity.

While the design of networks with higher throughput and lower delays has attracted a significant interest from the research community, design of practical tools that enable correct-by-construction networking systems has not received sufficient attention. Current implementations of network protocols rely on an ad hoc rather than a systematic approaches. There are little tools available for network engineer to design, implement, and verify the performance of reliable and robust protocol stack.

Using a modular approach, an implementation of a network protocol stack can be divided into four major components in general: message layer, state machine, system interface, and configurations, as shown in Figure 1.1 [1]. The message layer is responsible for conversion between the the bit stream received from a networking card and data structures residing in the memory on the same device. All other components use message layer as the underlying instrument to perform their functions. State machine keeps track of the protocol state based upon the sequence of messages received and processed. The configuration component provides parameters for protocol stack operation, such as identify numbers and network addresses. System interface specifies the functions through which the protocol interacts with the underlying machine environment. It helps facilitate the transmission of binaries that are being handled by the message layer.

A successful message layer implementation should be both safe and efficient. These characteristics not only prevent the exposure of potential vulnerabilities to the external world but lead to a better throughput of network devices. The quality of the message layer directly affects the other components because of their dependence on the message layer. In particular, state machine cannot successfully perform its duties without a solid underlying message layer. Since the message layer is a critical and complex component of the protocol stack, it is worthwhile to isolate it from the other parts of the protocol stack implementation in order to study methods to improve its performance and level of conformance.

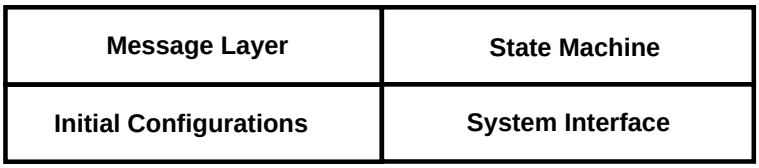


Figure 1.1: Major Components of a Network Protocol

It is challenging to have a conformant and efficient implementation, especially for protocols with complex message structures. As shown in Figure 1.2, mistakes and vulnerabilities exist in the implementations of some of the most widely used protocols by mature commercial companies and open source projects [1]. There are many open questions related to the implementation of the message layer of a network protocol, such as OpenFlow. The questions can be classified as one of the two major categories: *conformance* or *performance*. Some questions about conformance include: how do we know our implementation is being conformant to the protocol specifications? How do we design a verification process for the message layer that can be executed in a relatively short amount of time but yet still have a good coverage on potential vulnerabilities? How do we make the verification results can be used directly to eliminate those vulnerabilities? On the other hand, some questions regarding performance include: how do we characterize the performance of a message layer implementation? What are some major design choices and how do those choices affect safety and performance?

| Proto. | Age | Bug Date | Vendor | Error | CERT # |
|---|---|---|---|---|---|
| 802.11i | 2004 | 2012 | Broadcom | semantic | 160027 |
| OSPFv2 | 1998 | 2012 | Quagga | struct | 551715 |
| NTPD | 1985 | 2009 | GNU | struct | 853097 |
| ICMP | 1981 | 2007 | Cisco | both | 341288 |
| VTP | 1996 | 2006 | Cisco | semantic | 821420 |
| Bootp | 1985 | 2006 | Apple | struct | 776628 |

Figure 1.2: Sample Message Layer Vulnerabilities in Existing Implementations.

The objective of this thesis is to address these questions by using the message layer of OpenFlow as a case study. There are three major objectives for this thesis. The first one is to design and implement a correct-by-construction OpenFlow message layer. The second objective is to develop a verification framework that generates test cases to examine the level of conformance in several existing OpenFlow message layer implementations. Finally, the last one is to measure the performance impact of some design choices. With conformance and performance frameworks demonstrating the level of safety and efficiency, this work contributes to better understanding of design trade-offs and presents a systematic and disciplined approach for design and implementation of the message layer.

## 1.1 OpenFlow

OpenFlow is an application layer protocol that operates over TCP/TLS. As shown in Figure 1.3, the protocol separates the controller plane away from the networking devices in order to achieve a more centralized control on an otherwise distributed network. OpenFlow defines a set of common operations for controllers to configure switch states. The OpenFlow controllers and switch exchange control messages to carry out those operations.

Each OpenFlow message has an 8-byte header that is shared by all message types

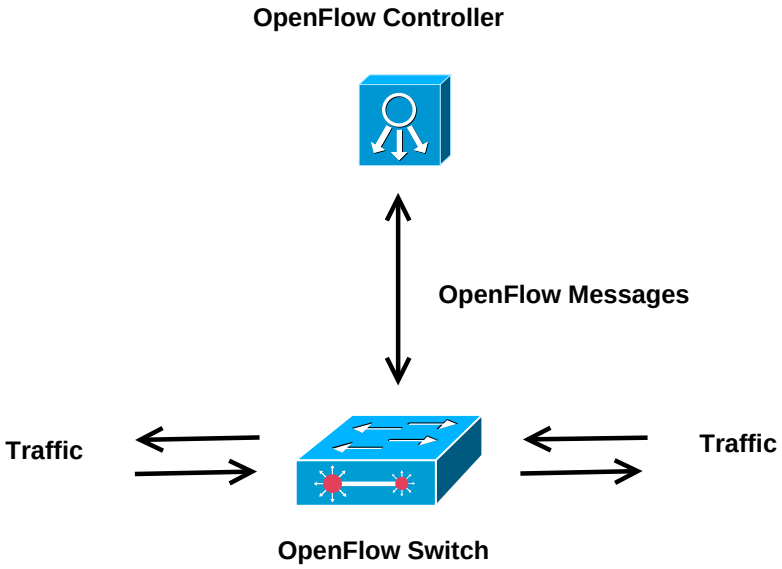


Figure 1.3: OpenFlow Controller and Switch

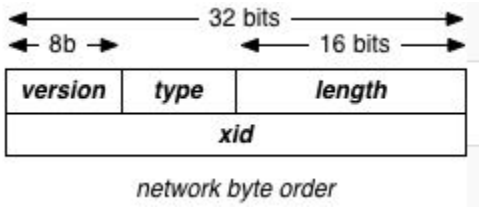


Figure 1.4: Structure for OpenFlow Header

from all released versions. As can be seen in Figure 1.4, the header specifies version, type, length and transaction ID of the message. The structure of header has not changed across all versions of OpenFlow. Several message types uses only the header as the entire body. One example is echo request and echo reply messages types, which are sent periodically to detect liveliness of the connection between the switch and the controller.

OpenFlow operates at the level of flow. All packets with the same flow signature belong to the same flow. Flow signatures are contained in a structure called Match. Figure 1.5 shows the structure for Match in version 1.0. Packets are classified into

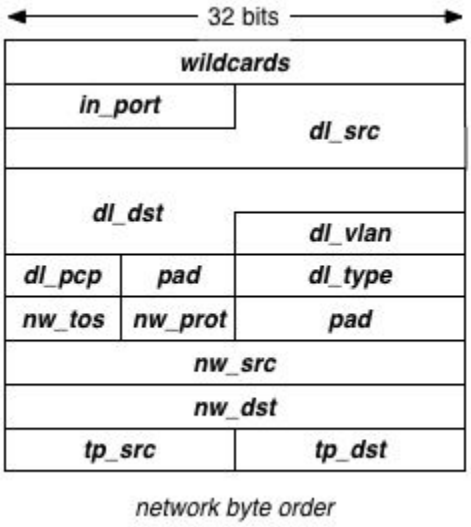


Figure 1.5: Structure for Match in OpenFlow 1.0

different flows based on their destination/source MAC, IP and port, among other packet information.

OpenFlow Actions specifies the policies on the flows. Some examples include forwarding the packet to a specific port (type Output) and inserting the packet into a particular queue in a packet (type Enqueue). Several policies can be applied on the same flow by attaching a vector of Actions with various types in the end of flow modification (Flow_mod) messages. The structures for Action's header and payload are shown in Figures 1.6.

Packet_in is a common message type issued by the switch to the controller. Its main function is to query the controller for the action(s) for an unknown flow that does not have an entry in the switch flow table. Its structure can be seen in Figure 1.7.

Flow_mod is sent by the controller to the switch in order to modify the flow table. Its structure can seen in Figure 1.8. Flow_mod is arguably the most important message type and it is sent from an OpenFlow controller to the switch in order to

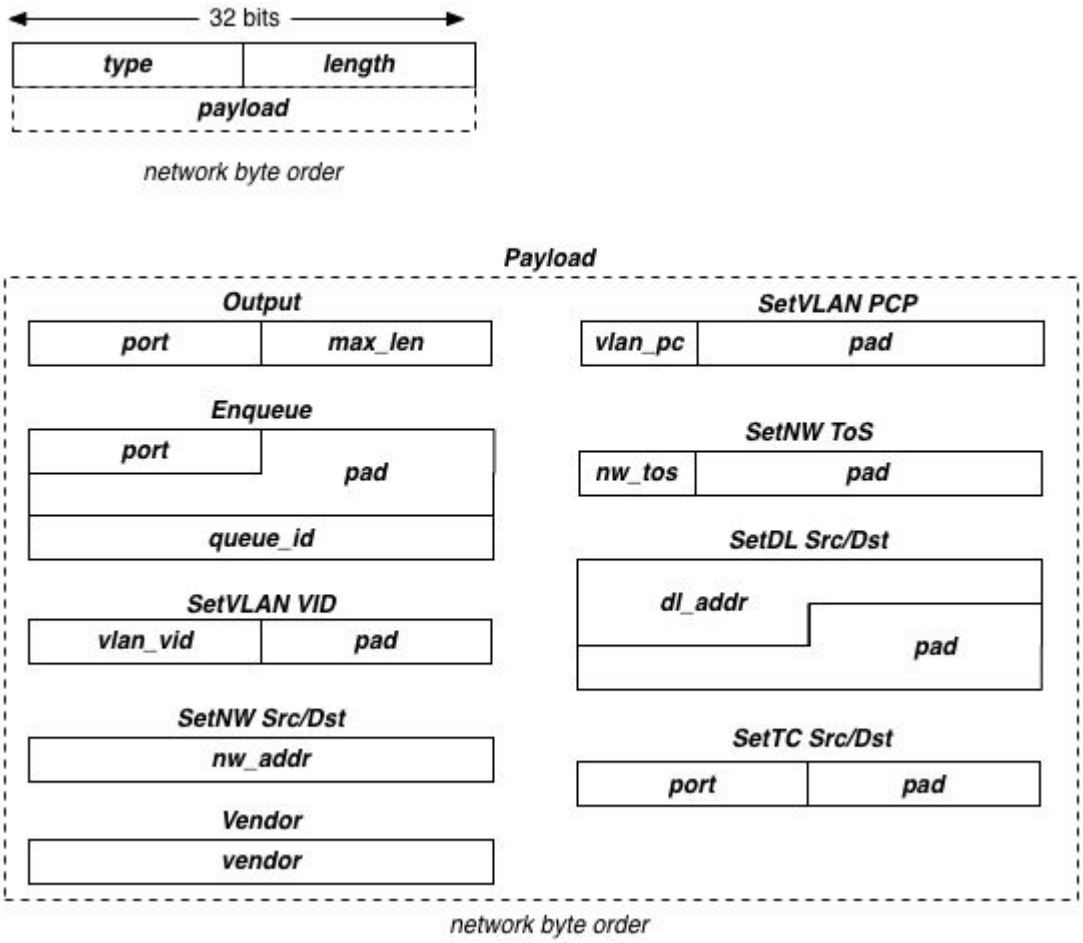


Figure 1.6: Message Structure for OpenFlow Action Header

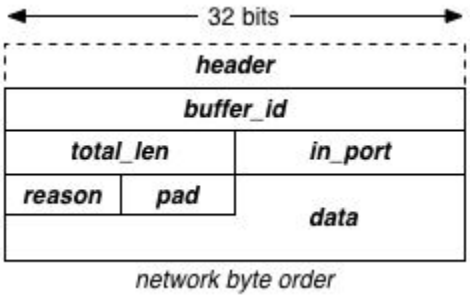


Figure 1.7: Message Structure for Packet_in in OpenFlow 1.0

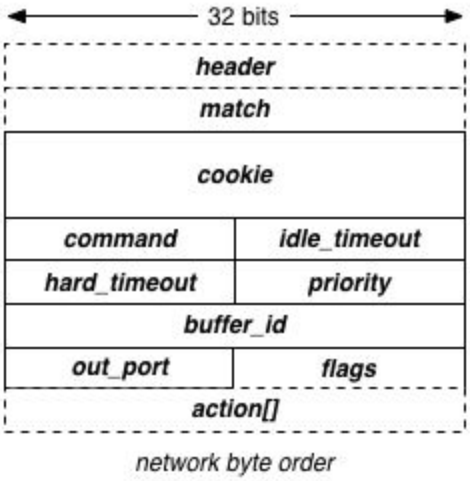


Figure 1.8: Message Structure for Flow_mod in OpenFlow 1.0

modify its flow table. It has a Match to classify the flows and a vector of Actions to define policies on these flows.

## 1.2 Message Layer

Message layer performs message construction and serialization. It has several major functions. As shown in Figure 1.9, the first one is to interpret the received raw binary streams. Depending upon the interpretation, messages with certain types need to be constructed using various data structures and stored into the memory on the current networking device. Going the opposite direction, the second major function is to turn the data structures in the memory that represent different types of messages into a binary stream in order to be transmitted through the system interface. Furthermore, the message layer is not only responsible for translating between the binaries from NIC and data structures that contain those messages in the device memory, but should also be able to determine the validity of incoming messages. The validation should successfully accept well-formed messages and reject ill-formed ones. Finally, it needs to coordinate with other modules in the stack

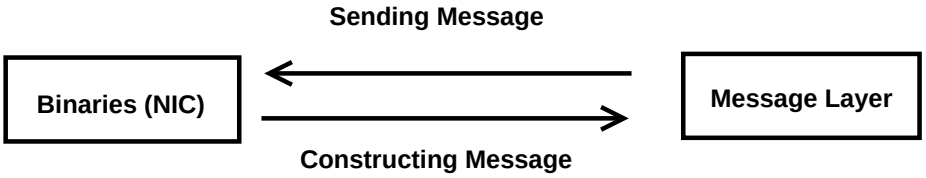


Figure 1.9: Message Layer in Network Protocols

implementation. The message layer is being used by other major components of the protocol. Thus, its ability to handle message safely and efficiently could affect the entire stack.

Similar to TCP and UDP, OpenFlow is a binary protocol. There is also a group of ASCII protocols such as HTTP. These ASCII protocols tend not to focus on performance because of the fact that they process text strings instead of binaries. However, even though OpenFlow is used as an example in this thesis, most of the underlying ideas of conformance verification and performance evaluation proposed in this work can be applied to both binary and ASCII protocols.

OpenFlow is an ideal protocol to be used as the example for message layer analysis since its message structure is the most complex part of the protocol. The number of types of messages in OpenFlow is much larger than protocols such as TCP. These numbers can be seen in Table 1.1. The number increases from 22 in version 1.0 to 33 in version 1.4. Furthermore, it has gained tremendous popularity since its release. This provides more existing open source implementations for examination of their conformance and/or performance.

## 1.3 Related Works

Existing tools tends to focus on the entire protocol. They take a black box approach to interact with the protocol implementation through its system interface. It would be helpful to gain more insights by using a more white box approach to

Table 1.1: OpenFlow Version vs. Number of Message Types.

| Version | # Msg Types |
|---|---|
| 1.0 | 22 |
| 1.1 | 24 |
| 1.2 | 26 |
| 1.3 | 30 |
| 1.4 | 33 |

inspect only the message layer itself.

There are several existing open source testing tools for OpenFlow. They can be catogorized into three major categories: functionality, performance, and applications. Table 1.2 demonstrates the classification. Cbench [2] is a controller benchmarking tool that measures the controller performance by simply injecting Packet_in messages. OFLOPS [3] is a switch benchmarking tool. Similar to Cbench, it takes a black box approach to test the performance. OFTest [4] and Test Specification performs functional testing on switches with some examples on message structures. SOFT [5] utilizes symbolic interpretation to find non-conformities between two implementations and to detect protocol bugs through invariant violations. In terms of application tools, NICE [6] and FlowChecker [7] are both concerned with finding inconsistencies in applications or configurations. VeriFlow [8] attempts to examine network-wide issues at real time. None of these tools focus on the conformance or performance specifically to the message layer of OpenFlow. Meanwhile, having a framework that better understands the conformance and performance at the message layer can help these tools narrrow down the reasons for obtained results.

Several works perform bounds testing in general software engineering, such as bounded exhaustive testing [9]. However, they have several limitations specific to the message layer of network protocols. They tend to be ineffective in dealing with

Table 1.2: Existing OpenFlow Tools

| | |
|---|---|
| Performance | Cbench, OFLOPS |
| Functionality | OFTest, Test Specification 1.0, SOFT |
| Application | NICE, FlowChecker, VeriFlow |

dependencies in the structures among different fields, especially type and length, which is prevalent in network protocols with type-length-value data structures. The second limitation of these bounds testing works is that they do not understand protocol semantics, such as checksum calculation in TCP.

In terms of actual test message generation, there are open source automated testing tools that conduct fuzzing on fields and potentially produce input tests for the message layer. However, most of them focus on value equality as opposed to representation equality. Other than the guarantee on the equality of values, representation equality guarantees the consistency in memory layout. For example, Figure 1.11 is an example with version (1 byte), type (1 byte) and length (2 bytes), which are the first three fields in the OpenFlow header, being read from the wire. However, if the testing tool only provides value equality, depending upon the implementation, the constructed structure in the memory can actually look like 1.10. They result in a different bit stream in the memory because of the padding that is generated in the process. A useful fuzzer for the message layer needs to have both value and representation.


Figure 1.10: Example of Value Equality

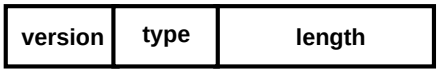


Figure 1.11: Example of Representation Equality

A small group of protocol fuzzers such as Sulley [10] also provide representational equality. However, similar to bounds testing, they have limitations on understanding protocol semantics. Thus, they do not possess the capability to fuzz over certain types of structures, such as a vector of Actions or Instructions in OpenFlow since each Action or Instruction can potentially have a different type.

Furthermore, there are other advantages of having a custom testing suite instead of randomized testing. For example, potential space for input messages is large for randomized testing messages produced by a protocol fuzzer. It would be computationally exhaustive to produce enough random test messages that have a good coverage on the message layer. A verification methodology is proposed in this thesis that can reduce the input space of testing messages. Messages generated with this methodology can also pinpoint the exact issues in the message layer.

Finally, PacketTypes [11] and Binpac [12] address message layer from a language perspective. They treat message formats as grammars of a domain specific language. They impose a learning curve for the users. Furthermore, even though they generate the source code, they do not generate test cases for verification.

# 2. DESIGN CHOICES

Selection of design choices can not only influence the level of conformance, but affect the performance of a message layer implementation as well. Understanding the consequences of these choices can provide developers a more systematic approach when it comes to implementing a message layer. The following is a sample of some of the design trade-offs [13].

**Degree of Validation**: Constraints on a message that determine its validity can be categorized as either structural or semantic. A structural constraint is violated if some value in the message contradicts what is known about the shape of the message—whether the reported content accurately reflects the data actually read. An example is a length field that indicates that a payload is larger than the message actually received. A semantic constraint is violated when some value within a message is not within the range of values required by the protocol specification, such as unsafe casting of message types. An implementation should not process messages with structural constraint violations, and it should be wary of semantic constraint violations. Attempting to process these messages exposes the stack or its client applications to potentially exploitable vulnerabilities.

**In-place Interpretation vs. Copy**: It is sometimes possible to construct a message simply by interpreting a buffer as some other type, applying byte-ordering transformations as required. Such an application of zero-copy can result in performance boost. On the other hand, copying can often provide an extra layer of protection on certain operations.

**Eager vs. Lazy Evaluation**: An example of eager vs. lazy can be semantic validation of the OpenFlow message layer. Any field describing the structure of the

message must be validated. Otherwise, the stack could easily misinterpret arbitrary data as meaningful, potentially leading to bugs and vulnerabilities. The validation can be performed proactively to detect issues early on or lazily once the message has been read into the memory.

**Time vs. Size**: A memory vs. time trade-off is where more memory is used to achieve a faster execution time. On the other hand, the increase in the use of memory can have negative impact on the performance. A potential reason can be that with fixed size caches, more memory usage can lead to more cache misses. These cache misses can come from both data and instruction caches.

An example of trade-off between memory and time can be inlining. A C/C++ implementation can rely heavily on inlining to improve performance at the expense of binary executable size. With inlining, the function bodies of the callees are directly copied into their callers. The same callees can be copied multiple times if they have multiple callers. On the other hand, the overhead of function call setup for small functions is reduced. This can potentially come at the expense of larger code size and increases instruction cache miss rates.

## 2.1 Flowgrammable

The OpenFlow message layer implementation in this work is part of the Flowgrammable OpenFlow stack. Flowgrammable [14], a Software-Defined Networking startup, participated in the OpenFlow driver competition hosted by the Open Network Foundation and was awarded as one of the finalists. The stack was designed using a principled programming approach to produce safe and efficient code. A C++11 program was developed to handle all message layer operations: construction, serialization, equality comparison, etc. The safety component ensures unsafe messages, i.e., messages that violate structural or semantic constraints, will be han-

dled appropriately.

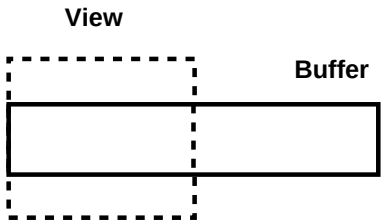


Figure 2.1: Buffer and View in Flowgrammable

During the implementation of Flowgrammable stack, design choices were explored. In-place interpretation is not used in the Flowgrammable stack in order to provide a guarantee of safe object construction. A constructed message object is separate from the buffer from which it was read. Figure 2.1 shows this separation. Figure 2.2 shows two major operations on the view: advance and constrain. When the current interpretation is finished, the buffer is advanced by creating a new view that has the next part of the message. On the other hand, constrain is used to limit the access of view on the buffer.

This mechanism of providing a view on the buffer is a potential source of performance degradation since it requires additional memory. However, the underlying principle is that conformance and safety must not be sacrificed for performance, and C++11 features such as move semantics are used to minimize the cost associated with these copies. Move constructors and assignments use rvalue references to minimize the overhead with decreasing the copying of temporary variables.

Furthremore, the Flowgrammable stack will lazily validate the non-structural fields of messages. The reason is that there are a number of useful applications (e.g., message filters) that do not access every field of a message. Requiring immediate val-

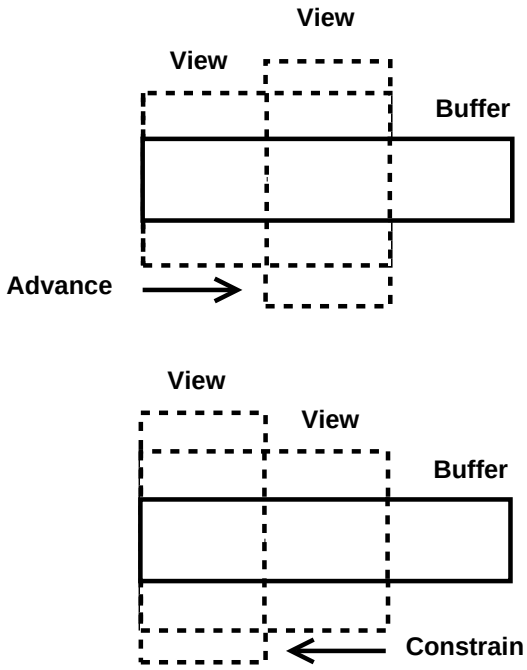


Figure 2.2: Operations on Buffer View in Flowgrammable

idation would be a poor use of resources for such applications. Flowgrammable also uses C++11's compile-time capability to evaluate constexpr extensively to reduce the runtime burden of computing message sizes and trivially validated constraints. Finally, some frequently used smaller functions are being inlined in order to achieve performance gain in terms of throughput.

# 3. FRAMEWORKS

The quality of message layer is vital to the quality of the entire OpenFlow stack since it is the most complex part of the protocol. For example, 70% of the 80K line-of-code in the Flowgrammable implementation is dedicated solely to the message layer. Furthermore, many of the other components in the stack, such as state machine and system interface all need to utilize the message layer to process message information. In order to gain confidence on the safety of an OpenFlow stack, its message layer needs to be extensively tested to discover potential vulnerabilities and performance inefficiencies. These issues can be resolved if the tests can pinpoint the exact location where failure or inefficiency occurs.

## 3.1 Conformance Verification

Most software testing fuzzers provide value equality. However, they typically do not provide representational equality (there are several exceptions), which means the layout of those values in memory might not be the same even though the equality of the values themselves are being guaranteed. Even with representational equality, another drawback with automated fuzzing is that very little confidence can be gained in each individual test case. Since the length field in the header is 2 bytes and any message, whose size is defined by this length, can be 65536 bytes long, the input space of possible testing messages becomes very large and infeasible to traverse through. There are several fields that only have a few fixed values, such as version. However, the small portion of those fields in the potential overall size does not help reduce the input space by much. It can take automatic fuzz testing a long time to reach to a satisfactory coverage. Thus, a heuristic is needed to reduce the input space size, and yet produce test cases that can give us more confidence.

As mentioned in the Degree of Validation section of Design, constraints on message layer can appear in two forms: structural and semantic. In the example from Figure 3.1, since the length field indicates the size of payload, its constraint (length no less than 8) is a structural constraint. On the other hand, constraints in the payload field are semantic constraints if no further component structures depend on it. Bounds are ending positions of a substructure in the memory (beginning and end of payload indicated by the length field). Stepping outside the bounds causes failures in interpreting the structure of the message. For example, in Figure 3.2, a length field slightly less than or larger than what is needed for the payload will result in a structural constraint violation. On the other hand, boundaries are the minimum and maximum values of a field. Stepping outside the boundaries will not affect the interpretation of the future messages from the network interface.

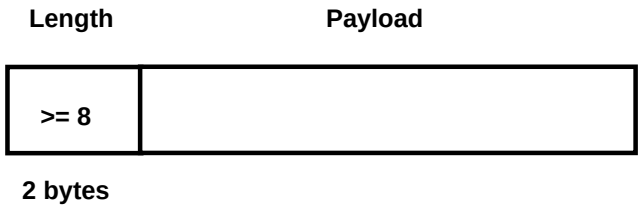


Figure 3.1: Illustration of Bounds and Boundaries

An example of stepping outside of bounds can be seen in Figure 3.2. The potential structural violations on this bound can stem from either direction.

A more concrete OpenFlow example can be an Flow_mod message in Version 1.0. The length of Flow_mod is: 64 + the length of Actions in the end. The size of the entire Flow_mod message is defined in the length field of OpenFlow header, which is affected by the size of each action that is specified in the length field of

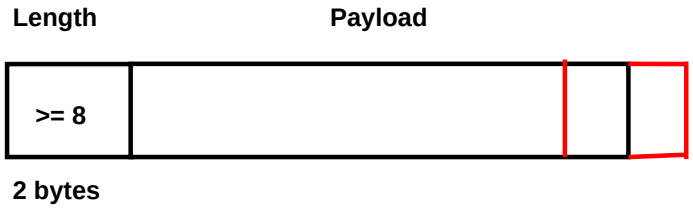


Figure 3.2: Illustration of Stepping Outside of Bounds and Boundaries

each action header. A length value that steps out of the bound (either longer or shorter) will violate structural constraints by failing to capture the payload. On the other hand, a flag value other than SendFlowRem, CheckOverlap or Emerg steps out of the boundary of this field and is a semantic constraint violation. A violation on this value will not cause failures in interpreting the remaining actions or subsequent messages.

Violations on conformance most likely occur at the bounds and boundaries. Thus, the strategy used in this work for conformance testing is to generate test messages that address all bounds and boundaries. Ill-formed messages are intentionally created to step out of the bounds (structural violations) or boundaries (semantic violations) for each message type and its substructures. For example, an invalid Flow_mod message is created with a length field of 65 + the length of its Actions to result in a structural failure. Another Flow_mod message is created with an invalid value in the flags field to cause a semantic failure. A conformant message layer should reject all these invalid messages while accepting and correctly interpreting valid messages that are free of either structural or semantic failures.

The hierarchy of OpenFlow messages can be modeled as a graph. All external nodes represent fields in a message and their parent internal nodes represent the data structures that group the children external nodes together. The call graph for version 1.0 that focuses on Flow_mod can be seen in Figure 3.3. The payload structures for all

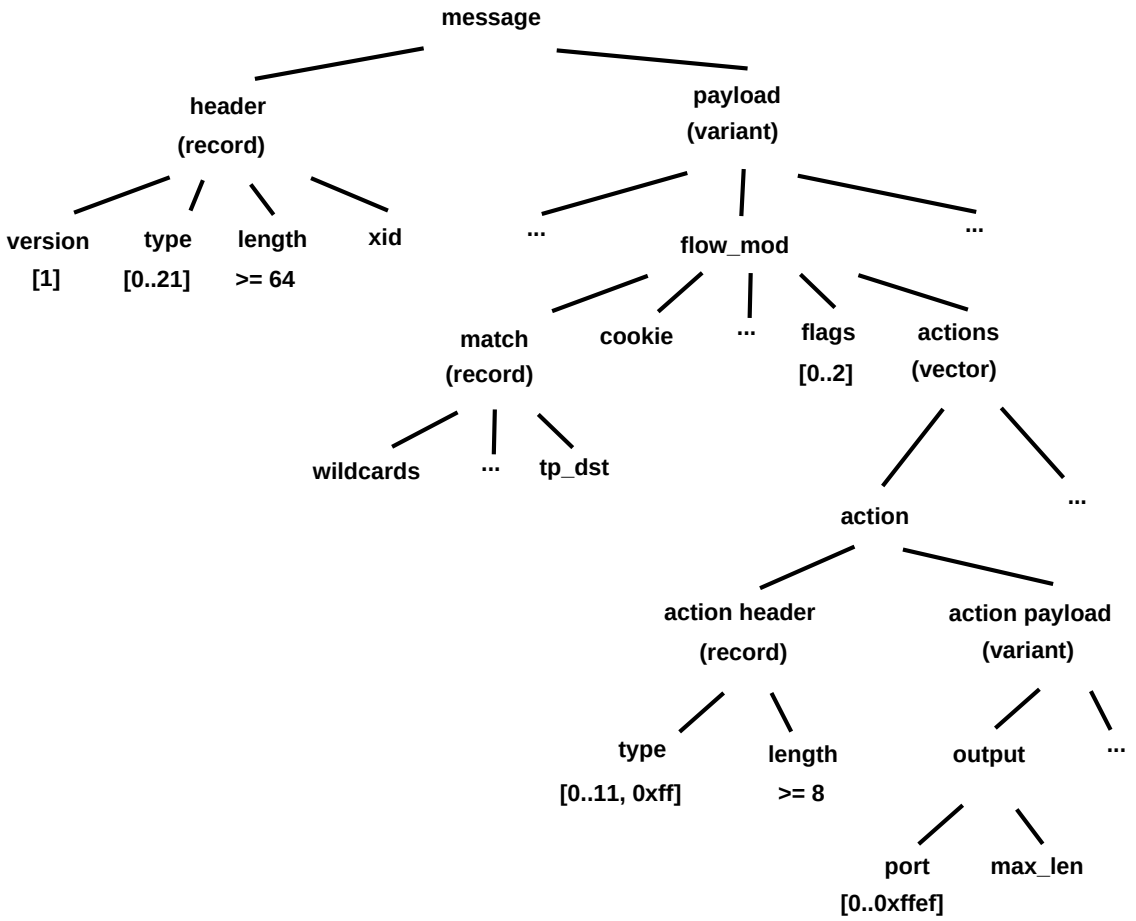


Figure 3.3: Call Graph of OpenFlow Message

other message types are being omitted in this graph due to space constraints. There are three major types of internal nodes in the graph: record, variant and vector. A record, such as Header and Match in version 1.0, only contains fixed size fields. A variant has several options on its structures, which is determined during runtime, often based upon the values in a type field in the earlier part of the message. A vector contains a natural number of the same type, such as uninterpreted data in the end of Packet_in, or a natural number of a variant with different types, such as actions embedded in the end of Flow_mod that specifies the policies for this particular flow.

There are many fields in the message layer of network protocols that has structural implications for other parts of the messages. The valid values for all external nodes in shown in Figure 3.3. These dependencies tend to be more than what general software testing tools can efficiently handle and require understanding on the semantics of protocol specifications. For example, in the case of OpenFlow, type often determines the choice for the structure of a variant during runtime. Sometimes a message, such as Statistics messages, requires multiples levels of variants to define its entire structure. Length also has structural dependencies with other structures, such as the number and types of actions in the end of Flow_mod. Each of these fields would have a range associated with it that specifies its bounds or boundaries. Each OpenFlow version has a graph representation of its own.

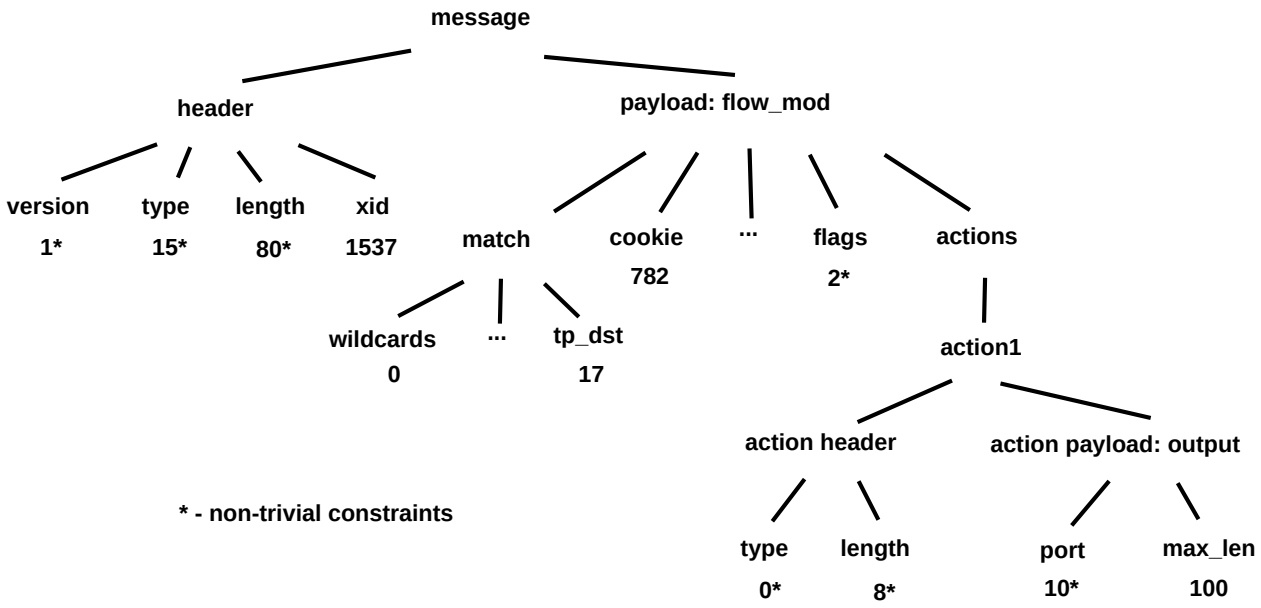


Figure 3.4: Example of OpenFlow Test Message Generation

The generation of OpenFlow test messages essentially is the preorder traversal of

the graph. The visits on each type in a variant will result in multiple test messages with each type has at least a positive and a negative message. An example message can be seen in Figure 3.4. It is constructed through traversing the graph from Figure 3.3. All external nodes in the graph with non-trivial constraints have been marked with a star. A value of 15 in the type field of record header determines that this is a Flow_mod message. It has an action with type output in the end, which is determined by a type value of 8 in the actions header. This action in the end also leaves the length field in the header with a value of 80. Because of the For each field that has either structural (e.g., type and length in the header) or semantic implications (e.g., flags), a negative message is generated by stepping right out of its bounds or boundaries in any direction. This way, each of these negative messages focuses on a particular issue in the message.

**Algorithm 1** Test Case Generation

```
procedure generateTests(none)
    traverse(root, null)
    initialize(goodMessageQueue)
    initialize(badMessageQueue)
end procedure
```

The starting point of the message generation process generation is shown in Algorithms 1, which is the beginning of the preorder traversal on message tree at the root. The other nodes will be visited in a recursive manner from here. Two message queues are also being initialized. They store the outputs for good messages and bad messages that target specific errors.

Algorithm 2 shows how the graph is being traversed in the processing of message generation. A message is being constructed each time the traversal reaches the end

**Algorithm 2** Traverse a Node in the Graph

```
procedure TRAVERSE(node, message)
    if node = null then
        processMessageEnd(message)
    end if
    if node is internal then
        processInternalNode(node)
    else if node is a leaf then
        processLeafNode(node)
    end if
    for child in children of node do
        traverse(child, message)
    end for
end procedure
```

of the tree. The handling of this part (processMessageEnd) will be shown later in Algorithm 3. Each node in the graph can either be an internal node (ex: record, variant, vector) or a leaf node. As described earlier, the children of internal nodes consist of leaf nodes, which represent fields in messages. Handling of internal nodes and leaf nodes are shown in Algorithms 4 and 5, respectively. Once the currently visited node is being processed, each child of the node will be visited in the next step.

**Algorithm 3** Process the End of a Message

```
procedure PROCESSMESSAGEEND(message)
    if isExisting(vector) then
        update(length)
    end if
    if isValid(message) then
        goodMessageQueue.enqueue(message)
    else
        badMessageQueue.enqueue(message)
    end if
end procedure
```

Algorithm 3 demonstrates the ending stage of a message generation. The valid flag is being checked to see if this is a good message in order to determine the appropriate output message queue it should be sent to. If vectors exist in the end, length fields that are related to the size of the vectors need to be updated. For example, in Figure 3.4, the length field is header is updated from 64 to 80 since there is a non-empty vector of Actions in the end of the Flow_mod message.

**Algorithm 4** Process an Internal Node

```
procedure processInternalNode(node)
    if node = variant then
        node.children ← structure(type.value)
    else if node = vector then
        node.addChild(empty)
        node.addChild(vector with max length)
        if vector of the same type then
            node.addChild(vector with multiple elements)
        else if vector of variants then
            node.addChild(vector with multiple variants)
        end if
    end if
end procedure
```

Algorithm 4 illustrates the processing of an internal node during tree traversal. Special processing is needed for a node that is either a variant or a vector. If the node is a variant, its branches are being trimmed and its only child is determined by the value of its type defined previously. If the node is a vector, several children are added to the node in order to achieve a good coverage.

Leaf node handling is shown in Algorithm 5. For each leaf node, if it has non-trivial structural or semantic constraints, messages are generated with values addressing all bounds or boundaries of this node. If the constraints for a field are

**Algorithm 5** Process a Leaf Node

```
procedure processLeafNode(node)
    if node has non-trivial constraints then
        for bound/boundary in bounds/boundaries of node do
            addMessages(node, message)
        end for
    else
        for i ← 1,n do
            message.append(random)
            visitNextNode(node, message)
        end for
    end if
end procedure
```

trivial, meaning it will take any value to be valid, a random value will be used to advance the buffer in the message generation. A user input can be used to indicate how many times a random value needs to be selected to fill a field with trivial constraints. This will determine the size of the test message suite. A message is completed when there are no leaf nodes left in the graph.

Algorithm 6 shows the addition of messages for a leaf node with non-trivial constraints. There are two types of non-trivial constraints: range constraints or named constraints. Range constrains represent a set of valid values as long as they belong in a specific range. Its bounds or boundaries are the maximum and minimum values of the range. Some range constraints, such as length, only has one valid value. Named constraints have an enumeration of valid values, all of which are bounds/boundaries. Some examples include type and special port values (e.g., Controller). If the partially constructed message is already an invalid message, since it is carefully designed to fail at another point, only valid values will be filled in the remaining parts of the message. On the other hand, for a message that is valid up to this point, new messages are created with valid and invalid values for the currently visited node before

**Algorithm 6** Add Messages at a Leaf Node

```
procedure addMessages(node, message)
    if isValid(message) then
      newGoodMessage ← message
      newGoodMessage.append(node.validBound/Boundary)
      visitNextNode(node, newGoodMessage)
      newBadMessage ← message
      newBadMessage.append(node.invalidBound/Boundary)
      newBadMessage.valid ← false
      visitNextNode(node, newBadMessage)
    else      // no other failures
      newMessage ← message
      newMessage.append(random)
      visitNextNode(node, newMessage)
    end if
end procedure
```

the traversal procedure advances to the next node. Each invalid value for the current node should start stepping out of a bound or a boundary, depending on whether the constraint is structural or semantic.

The generation of test message suite described above has no dependencies with the type of applications the message layer will experience. Several types of variations can be applied to the process of message suite generation. A possible variation of this generation procedure can be a probabilistic adjustment to the type of traffic. For example, type in the header can be weighted to include a higher percentage of Flow_mod and Packet_in messages since these two messages tend to be used frequently between the controller and the switch.

## 3.2 Performance Evaluation

The performance evaluation framework uses regular metrics including: dynamics instruction count, cycles, branch prediction misses, cache misses, and throughput. Existing tools are being used. The profiling tool Perf [15] is used to measure the

performance of OpenFlow devices under test. It uses sampling and aggregates the final results through these samples. There is a sweet spot for the sampling period. Undersampling does not provide resolution for an accurate final output. On the other hand, oversampling can negative impact the results due to sampling overhead. The ideal sampling period is found to be around 1ms. Some of the benchmarking parameters include instructions, branch predictions and overall throughput. Several other tools were used in the measurements. Valgrind [16] is used for some memory/cache measurements, such as instruction cache and heap usage.

# 4. EXPERIMENTAL SETUP

In order to evaluate the efficacy of our approach to the design and implementation of the message layer, a benchmarking framework is being constructed to support the comparison of message layers in terms of protocol conformance and performance. The framework consists of messages generated from the methodology described in the Conformance Verification section. The test cases contain both conforming and non-conforming messages, and the latter set contains messages with both structural and semantic constraint violations. These test cases cover all bounds and boundaries of OpenFlow message types and their substructures.

Other than Flowgrammable, the OpenFlow devices under test include Beacon [17], CPqD Soft Switch versions 1.2 and 1.3 [18]. Beacon controller is a leading SDN stack and Soft Switch is developed by CPqD, winner of the OpenFlow driver competition. All the experiments are conducted on a Dell PowerEdge 720 server (courtesy of Texas A&M IT Department). The server has an Intel Xeon E5 2637 processor and 8GB of RAM. The experiments are run on Ubuntu 12.04. The compiler for C/C++ implementations (for Flowgrammable and Soft Switch) is gcc 4.8.1 with optimization -O2. Java runtime (for Beacon) is OpenJRE with Java 6 update 27 based on IcedTea6 version 1.12.6.

Figure 4.1 shows the experimental setup. For each device that is under test, its message layer source code is carefully examined and extracted from the entire source code. The extracted code contains parts necessary to perform message layer functions. The message layer testing constructs OpenFlow messages object for each message in the test suite. For conformance verification, it records whether the constructions report success or failure against the expected results (positive and negative

tests). For performance evaluation, it collects the output for measured metrics.

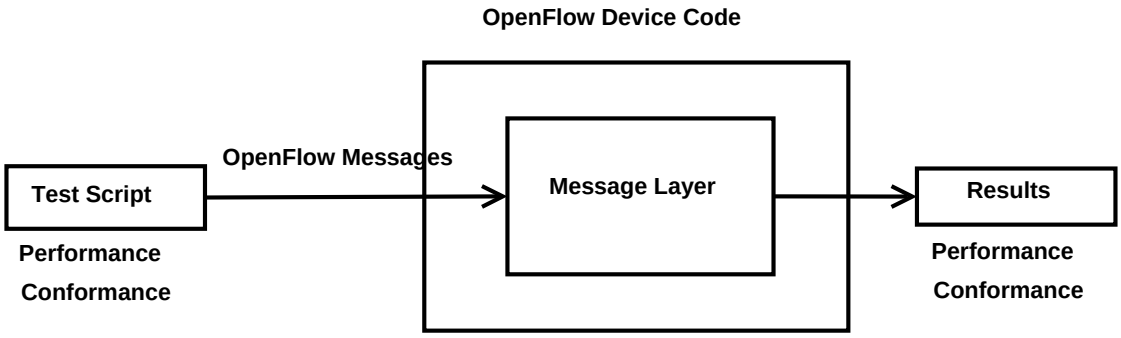


Figure 4.1: Experimental Setup

# 5. RESULTS

This chapter describes data collected from the experiments. It includes results for test generation, conformance verification and performance evaluation. The test cases produced during test generation are used in both conformance and performance experiments.

## 5.1 Test Cases

Using methodology described in the previous section, the generated test cases for both structural and semantic testing from version 1.0 to version 1.3.1 can be seen in Table 5.1. Later versions tend to have more test cases because of the increase in the number of types and their complexities. It should also be noted that the number of testing messages that targets structural constraints is much larger than that of semantic constraints. The reason is that most of the constraints on the fields have structural implications for other parts of the message. These messages are used to prevent structural and semantic violations in messages. Each negative message exposes a particular failure in the interpretation.

Table 5.1: Distribution of Generated Test Messages for Different Versions.

| | 1.0 | 1.1 | 1.2 | 1.3 | 1.3.1 | Total |
|---|---|---|---|---|---|---|
| Positive | 106 | 186 | 278 | 291 | 290 | 1151 |
| Negative Structural | 137 | 235 | 423 | 633 | 633 | 2061 |
| Negative Semantic | 21 | 24 | 33 | 33 | 39 | 150 |

## 5.2 Performance Evaluation

Excluding semantic checking gives the stack the ability to accept soft failures when it does not affect the processing of other structures. This can help improve performance in certain situations. Table 5.2 shows the overhead of semantic checking in the Flowgrammable stack. The overhead on the throughput is minimal. Major statistics such as instruction count, cycle count and branch misses do not change with or without semantic checking. This shows that the accuracy does not need to be traded for time in Flowgrammable stack. However, this minimal difference in throughput here can be larger with other types of OpenFlow traffic or with other implementations for other network protocols. It would still be interesting to include an evaluation for semantic checking overhead in order to determine whether it is worthwhile to trade accuracy for time in certain situations.

Table 5.2: Semantic Checking Overhead on Flowgrammable Message Layer.

| | w/o Semantic | w Semantic |
|---|---|---|
| Dynamic instruction count | 51,059,684,477 | 51,075,452,267 |
| Cycle count | 32,724,713,505 | 32,811,718,428 |
| IPC | 1.56 | 1.56 |
| Branch count | 11,437,138,342 | 11,440,761,046 |
| Branch mispredictions | 25,361,867 | 25,888,803 |
| Branch misprediction rate | 0.22% | 0.23% |
| Execution time | 9.89s | 9.93s |
| Avg msg/s | 53,380 | 53,184 |
| Avg throughput | 2.25Gbps | 2.24Gbps |
| Avg msg latency | 19$\mu$s | 19$\mu$s |

Inlining smaller functions can increase code size due to the multiples copies of the body of those functions. This might cause an increase in instruction cache misses due to the increase in static instructions. On the other hand, direct copying of function bodies can increase the average number of instructions in a branch, which tends to have fewer branch prediction misses. However, such a trade-off between instruction cache miss and branch prediction miss is not occurring in the Flowgrammable stack. The reason is that the code size actually decreased with inlining. Such a phenomenon is caused by the fact that most inlined functions belong to a specific message type and they are only being copied once (ex: message byte calculation). The number of static instructions decreases by about 1% (Table 5.3) with fewer callq and retq instructions. Coupling with the fact that an increase of approximately 8% in instructions per branch (4.64 vs 4.30), inlining increases the throughput for about 5%.

Even though inlining in this case does not involve a trade-off between instruction cache usage and branch prediction, it can still occur in other situations. Too much inlining may even hurt the overall performance. It would be useful to evaluate the impact of inlining on the throughput to determine whether a trade-off between memory and speed is beneficial.

Table 5.3: Inlining Overhead on Flowgrammable Message Layer.

| | w/o Inlining | w Inlining |
|---|---|---|
| Dynamic instruction count | 53,574,927,228 | 51,075,452,267 |
| Static instruction count | 249,017 | 246,434 |
| Cycle count | 35,046,075,476 | 32,811,718,428 |
| IPC | 1.53 | 1.56 |
| Branch count | 12,349,371,439 | 11,440,761,046 |
| Branch mispredictions | 26,653,382 | 25,888,803 |
| Branch misprediction rate | 0.22% | 0.23% |
| Instructions per branch | 4.34 | 4.64 |
| Instruction cache miss | 82,522,436 | 82,014,764 |
| Execution time | 10.47s | 9.93s |
| Avg msg/s | 50,446 | 53,184 |
| Avg throughput | 2.13Gbps | 2.24Gbps |
| Avg msg latency | 20$\mu$s | 19$\mu$s |
| Code size | 1.53MB | 1.49MB |
| Heap usage | 4,411,015,389 bytes | 4,411,015,389 bytes |

Optimization on common operations can be explored for performance improvement. Header processing is a type of common operations in message layer since every packet will have a header. Each time the Beacon controller receives an OpenFlow message from an OpenFlow-capable controller, its OpenFlow stack implementation module reads and interpret the header of each message twice. The first time it determines message type and whether the buffer has enough space for a message whose size is specified the length field. Then, it instantiates the payload object and

allocates memory for the structure based upon the type. Afterwards, it moves the pointer in the buffer back to the beginning of the message. The potential inefficiency here lies in some unnecessary extra processing for the first header interpretation, during which transaction ID is not being used by the interpretation. Thus, a potential optimization on Beacon can be skipping transaction ID during the first header interpretation.

Test messages that are about 65KB in length are not being included in this experiment in order not to virtually eliminate header overhead due to large packet size. After excluding those long messages, t he average packet size decreases to about 136 bytes. The results show that optimizing the header improves performance by about 3% (Table 5.4), which corresponding to the size of transaction ID (4 bytes) in the average packet size. Even though this particular example of header optimization does not improve the performance drastically because of the header length and message size, header processing optimization could still substantially improve performance in many other cases.

Table 5.4: Beacon Header Processing Optimization.

| | w/o Optimization | w Optimization |
|---|---|---|
| Dynamic ins count | 38,409,868,843 | 37,748,135,092 |
| Cycle count | 33,817,454,266 | 33,779,591,010 |
| IPC | 0.88 | 0.89 |
| Branch count | 6,174,352,333 | 6,169,341,454 |
| Branch mispredictions | 138,487,422 | 137,983,068 |
| Branch misprediction rate | 2.24% | 2.24% |
| Execution time | 10.69s | 10.41s |
| Avg msg/s | 90,513 | 92,990 |
| Avg throughput | 1.00Gbps | 1.02Gbps |
| Avg msg latency | 11$\mu$s | 11$\mu$s |

## 5.3 Conformance Verification

Table 5.5: Conformance Verification Results.

| | Beacon 1.0 | Soft Switch 1.2 | Soft Switch 1.3 | Flowgrammable |
|---|---|---|---|---|
| Structural failures | 3.7% | 31.8% | 38.1% | 0% |
| Semantic failures | 28.8% | 15.7% | 47.8% | 0% |

The passing percentage of conformance verification test cases for different stacks can be seen in Table 5.5: Most of the failures in Beacon are the result of semantic violations. For example, the value of reason field in Packet_in, Flow_removed,

Port_status messages; the value of command filed in Flow_mod; the flag field in Flow_mod, Get_config_res, and Set_config are not being verified by the message layer of Beacon controller's OpenFlow stack.

Most of the failures in Soft Switch come from the fact that it does not reject messages with shorter lengths than in the buffer in hope that the extra bits will be carried into the next message. Some other failed test cases include invalid length of table feature. Table feature is a type of Statistics query on the flow table. Unlike other parts of the message in which length field is included as part of the value specified by it, table feature's length is not inclusive of it.

# 6. CONCLUSION AND FUTURE WORK

Flowgrammable produced a strong and conformant message layer that helps award them the finalist for the OpenFlow driver competition. It creates a conformance verification framework that generates test cases covering bounds and boundaries of all message types. The same conformance framework is being used to evaluate and discover issues in several other OpenFlow stack implementations. Some design choices in Flowgrammable and Beacon are being explored. Their performances are measured to observe the impact of those design choices. Even though some design choices do not lead to a large performance difference in this case, understanding them will still help developers achieve more efficient and safer implementations of the message layer of networking protocols in general.

The work can be extended in many different directions. A tool to synthesize and automate the implementation can also help provide a unified interface to conformance and performance frameworks. Some other design choices can be explored. For example, the effects of shifting computation in time, such as lazily evaluating the length of a message, can be studied. A tool that can synthesize and automate the message layer will be able to help facilitate this since time consuming manual modification of the implementation can be minimized. The same analysis on conformance and performance can be conducted on other OpenFlow stack implementations such as Floodlight controller [19], Ryu controller [20], and Open vSwitch [21]. The same procedures can also be repeated on other network protocols. Traffic through test bed in more realistic scenarios can help study the impact of design choices on performance in various networking applications. This can be done by building a physical OpenFlow network or in a virtual network such as GENI [22]. Finally, hardware

acceleration can be taken into consideration. The main goal will be to identify the bottle neck in the instruction level, meaning recognizing the most frequently used instructions during construction, serialization and resource management in the message layer. These bottlenecks can be then tackled with specialized hardware accelerators.

# 7. REFERENCES


[1] J. Casey, A. Sutton, G. Dos Reis, and A. Sprintson. Eliminating Network Protocol Vulnerabilities Through Abstraction and Systems Language Design. In *Proceedings of the 21st international conference on Network protocols*, ICNP '13, pages 1–6, Piscataway, NJ, USA, 2013. IEEE.

[2] Cbench - an OpenFlow controller bench-marker. Retrieved January 18, 2014 from `http://archive.openflow.org/wk/index.php/Oflops`.

[3] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. Moore. Oflops: An Open Framework for OpenFlow Switch Evaluation. In *Proceedings of the 13th international conference on Passive and active Measurement*, PAM '12, pages 85–95, 2012.

[4] OFTest - framework and test suite for the OpenFlow. Retrieved January 18, 2014 from `http://www.projectfloodlight.org/oftest/`.

[5] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A SOFT Way for OpenFlow Switch Interoperability Testing. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 265–276, New York, NY, USA, 2012. ACM.

[6] M. Caninsi, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI '12, pages 10–19, New York, NY, USA, 2012. ACM.

[7] E. Al-Shaer and S. Al-Haj. FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, SafeConfig '10, pages 37–44, New York, NY, USA, 2010. ACM.

[8] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. Veriflow: Verifying Network-Wide Invariants in Real Time. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 49–54, New York, NY, USA, 2012. ACM.

[9] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software Assurance by Bounded Exhaustive Tesing. In *Proceedings of the ACM SIGCOMM International Symposium on Software Testing and Analysis*, ISSTA '04, pages 133–142, New York, NY, USA, 2004. ACM.

[10] Sulley - A Pure Python Fully Automated and Unattended Fuzzing Framework. Retrieved January 18, 2014 from `https://github.com/OpenRCE/sulley/`.

[11] P. McCann and S. Chandra. Packet Types: Abstract Specification of Network Protocol Messages. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 321–333, New York, NY, USA, 2000. ACM.

[12] R. Pang, V. Paxson, R. Sommer, and L.Peterson. Binpac: A Yacc for Writing Application Protocol Parsers. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 289–300, New York, NY, USA, 2006. ACM.

[13] G. Varghese. *Network algorithmics*. Morgan Kaufmann, Burlington, Massachusetts, USA, 2010.

[14] Flowgrammable - An Openflow SDN Stack. Retrieved January 18, 2014 from http://flowgrammable.org.

[15] Perf - Linux profiling with performance counters. Retrieved January 18, 2014 from https://perf.wiki.kernel.org/index.php/Main_Page.

[16] Valgrind - Instrumentation Framework for Building Dynamic Analysis Tools. Retrieved March 17th, 2014 from http://www.valgrind.org/.

[17] D. Erickson. The Beacon OpenFlow Controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM.

[18] Soft Switch. Retrieved January 18, 2014 from https://github.com/CPqD/ofsoftswitch13.

[19] Floodlight - Open SDN Controller. Retrieved January 18, 2014 from http://osrg.github.io/ryu/.

[20] Ryu - SDN Framework. Retrieved January 18, 2014 from http://osrg.github.io/ryu/.

[21] Open vSwitch - an Open Virtual Switch. Retrieved January 18, 2014 from http://openvswitch.org/.

[22] Geni - Global Environment for Network Innovations. Retrieved March 17th, 2014 from https://www.geni.net/.